

LLVM based Program Pruning against Special Concerns

Hongxu Chen

December 6, 2013

LLVM based Program Pruning against Special Concerns

Hongxu Chen

December 6, 2013

Outline

1 Motivation

2 Approach

3 Issues and Solutions

- Accurately pruning unrelated snippets w.r.t concerns can:

Motivation

- Accurately pruning unrelated snippets w.r.t concerns can:
 - ① help program understanding

Motivation

- Accurately pruning unrelated snippets w.r.t concerns can:
 - ① help program understanding
 - ② help to generate regression testing

Motivation

- Accurately pruning unrelated snippets w.r.t concerns can:
 - ① help program understanding
 - ② help to generate regression testing
 - ③ **avoid unnecessary path constraints solving in symbolic execution**

Motivation

- Accurately pruning unrelated snippets w.r.t concerns can:
 - ① help program understanding
 - ② help to generate regression testing
 - ③ avoid unnecessary path constraints solving in symbolic execution
 - 👉 Focus on a specific assert error

- Accurately pruning unrelated snippets w.r.t concerns can:
 - ① help program understanding
 - ② help to generate regression testing
 - ③ avoid unnecessary path constraints solving in symbolic execution
 - 👉 Focus on a specific **assert** error
 - 👉 **Less code \Rightarrow less time to locate a special bug**

Motivation

- Accurately pruning unrelated snippets w.r.t concerns can:
 - ① help program understanding
 - ② help to generate regression testing
 - ③ avoid unnecessary path constraints solving in symbolic execution
 - 👉 Focus on a specific **assert** error
 - 👉 Less code \Rightarrow less time to locate a special bug
- For a given **program point** and an **entry function**:

Motivation

- Accurately pruning unrelated snippets w.r.t concerns can:
 - ① help program understanding
 - ② help to generate regression testing
 - ③ avoid unnecessary path constraints solving in symbolic execution
 - 👉 Focus on a specific **assert** error
 - 👉 Less code \Rightarrow less time to locate a special bug
- For a given **program point** and an **entry function**:
 - **find** **emph** **possible** **paths** **that** **reaches** **the** **point**

- Accurately pruning unrelated snippets w.r.t concerns can:
 - ① help program understanding
 - ② help to generate regression testing
 - ③ avoid unnecessary path constraints solving in symbolic execution
 - 👉 Focus on a specific **assert** error
 - 👉 Less code \Rightarrow less time to locate a special bug
- For a given **program point** and an **entry function**:
 - find emph possible paths that **reaches** the point
 - **get the statements affecting the point**

Outline

1 Motivation

2 Approach

3 Issues and Solutions

Requirements

- Pruning should be conservative

Requirements

- Pruning should be **conservative**
- Prune as many instructions as possible

Requirements

- Pruning should be **conservative**
- Prune as many instructions as possible
- The remaining snippets should be **executable**

Approach

- ① Build points-to set for the given translation unit \Leftarrow program point
- ② Build accurate callgraph \Leftarrow points-to
- ③ Eliminate un-called functions \Leftarrow entry function, callgraph
- ④ Build modification info \Leftarrow points-to
- ⑤ Inter-procedural Reachability Analysis \Leftarrow un-called function elimination
- ⑥ Inter-procedural Slicing \Leftarrow un-called function elimination, modify set, callgraph, points-to

Outline

1 Motivation

2 Approach

3 Issues and Solutions

Andersen's Inaccuracy

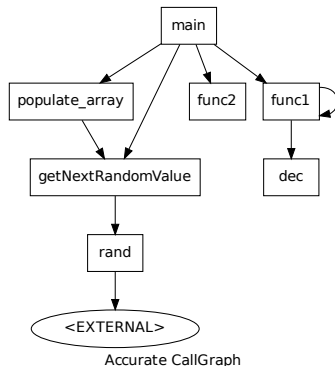
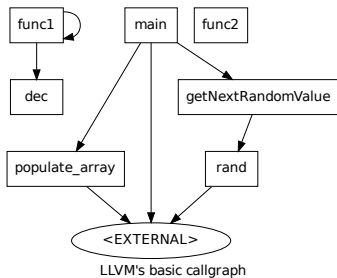
```
1      void foo(void) {
2          int a, i, j, k;
3          int *p;
4          if (a < 0) {
5              p = &i;
6              *p = 3;
7          } else if (a > 0) {
8              p = &j;
9              *p = 3;
10             assert(j > 0); // <= interest point
11         }
12         p = &k;
13         *p = 3;
14     }
```

CallGraph Example

```
1  int dec(int i) {
2      return i - 1;
3  }
4
5  unsigned long func1(int i) {
6      if (i == 0) return 1;
7      return func1(dec(i)) * i;
8  }
9
10 unsigned long func2(int i) {
11     return i * 0;
12 }
13
14 unsigned long (*pF)(int) = func1;
15
16 int getNextRandomValue(void) {
17     return rand() % 10;
18 }
```

```
1  void populate_array(int *array,
2      size_t arraySize,
3      int (*getNextValue)(void)) {
4      for (unsigned i = 0;
5          i < arraySize; i++)
6          array[i] = getNextValue();
7  }
8
9  int main(void) {
10     int i = 3;
11     int myarray[10];
12     if (i < 3) {
13         pF = func1;
14     } else {
15         pF = func2;
16     }
17     pF(getNextRandomValue());
18     populate_array(myarray, 10,
19         getNextRandomValue());
20 }
```

CallGraph Comparison



Inter-procedural Reachability

```
int foo(int i) {  
    if (i < 0) {  
        assert(0); // <=  
        return -i;  
    } else {  
        return i;  
    }  
}  
  
int foo1(int i) { return foo(i); }  
int foo2(int i) { return foo(i); }  
  
int main(void) {  
    int i = -1;  
    // foo1(i);  
    foo2(-i);  
    foo(i);  
    return 0;  
}
```

```
define i32 @foo(i32 %i) [  
entry:  
    %cmp = icmp slt i32 %i, 0  
    br i1 %cmp, label %if.then, label %if.else  
  
if.then:  
    call void @__assert_fail(<...>)  
    unreachable  
  
if.else:  
    ret i32 %i /// <=  
  
declare void @__assert_fail(i8*, i8*, i32, i8*)  
  
define i32 @foo1(i32 %i) [  
entry:  
    %call = call i32 @foo(i32 %i)  
    ret i32 %call  
]  
  
define i32 @foo2(i32 %i) [  
entry:  
    %call = call i32 @foo(i32 %i)  
    ret i32 %call /// <=  
]  
  
define i32 @main() [  
entry:  
    %call = call i32 @foo2(i32 0)  
    %call1 = call i32 @foo(i32 0)  
    ret i32 0 /// <=  
]
```

Thank You 😊

Thank You 😊

Thank You 😊

Thank You 😊

Thank You 😊